# Neural Network and Decision Boundary Visualization: Complete Deep Dive

## Table of Contents


1. **Introduction - What is a Neural Network**
2. **Value Class - The Magic of Automatic Differentiation**
3. **Neural Network Architecture - The Structure of Intelligence**
4. **Make_Moons Dataset - Understanding Data Patterns**
5. **Training Process - How Networks Learn**
6. **Decision Boundary Visualization - Seeing AI Decisions**
7. **Complete Code Workflow**
8. **Practical Insights and Tips**


## 1. Introduction - What is a Neural Network

A **Neural Network** is an artificial intelligence system that works like the human brain. It contains many **neurons** (nodes) that are interconnected and process data to recognize patterns and make decisions.

### Key Concepts:

- **Input Layer**: Receives the data
- **Hidden Layers**: Performs processing and pattern recognition
- **Output Layer**: Makes final predictions/decisions
- **Weights & Biases**: Network parameters that get adjusted during learning

**Real-life Example**: Just as humans can identify a dog or cat in a photograph, neural networks can also recognize patterns in data.

[101]

## 2. Value Class - The Magic of Automatic Differentiation

```
class Value:
    def __init__(self, data, _childeren=(), _op='', label=''):
        self._prev = set(_childeren)
        self.grad = 0.0
        self._op = _op
        self.label = label
```

```
        self._backward = lambda: None
        self.data = data
```

### Purpose of Value Class:

This class implements **Automatic Differentiation**. This means whenever any mathematical operation occurs, its derivative is automatically calculated as well.

[102]

### Key Components Explanation:

**1.** `self.data`: The actual numerical value
**2.** `self.grad`: The gradient (derivative) value
**3.** `self._prev`: Parent nodes (nodes from which this value came)
**4.** `self._op`: The operation used (+, *, tanh, etc.)
**5.** `self._backward`: Function that calculates the gradient

[103]

### Mathematical Operations:

**Addition (`__add__`)**:

```
def __add__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data + other.data, (self, other), '+')

    def _backward():
        self.grad += 1.0 * out.grad
        other.grad += 1.0 * out.grad

    out._backward = _backward
    return out
```

**Chain Rule Intuition**: When `c = a + b`, then:

- `dc/da = 1` (derivative of a is 1)
- `dc/db = 1` (derivative of b is also 1)

**Multiplication (`__mul__`)**:

```
def __mul__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data * other.data, (self, other), "*")

    def _backward():
        self.grad += other.data * out.grad
        other.grad += self.data * out.grad
```

```
    out._backward = _backward
    return out
```

**Chain Rule**: When `c = a * b`, then:

- `dc/da = b` (coefficient of a is b)
- `dc/db = a` (coefficient of b is a)

**Tanh Activation**:

```
def tanh(self):
    x = self.data
    t = (math.exp(2*x) - 1) / (math.exp(2*x) + 1)
    out = Value(t, (self,), 'tanh')

    def _backward():
        self.grad += (1 - t*t) * out.grad

    out._backward = _backward
    return out
```

**Derivative of Tanh**: `d/dx(tanh(x)) = 1 - tanh²(x)`

### Backward Propagation:

```
def backward(self):
    topo = []
    visited = set()

    def build_topo(v):
        if v not in visited:
            visited.add(v)
            for child in v._prev:
                build_topo(child)
            topo.append(v)

    build_topo(self)
    self.grad = 1.0

    for node in reversed(topo):
        node._backward()
```

**Topological Sort**: This ensures that gradients are calculated in the proper order.

### 3. Neural Network Architecture - The Structure of Intelligence

```
class Neural_Network:
    def __init__(self, input_size, layer_sizes):
        self.layers = []
        sizes = [input_size] + list(layer_sizes)  # sizes = [2, 4, 1]
```

```
        for nin, nout in zip(sizes, sizes[1:]):
            layer = []
            for _ in range(nout):
                weights = [Value(np.random.uniform(-1, 1)) for _ in range(nin)]
                bias = Value(np.random.uniform(-1, 1))
                layer.append((weights, bias))
            self.layers.append(layer)
```

### Architecture Breakdown:

**Input Size**: 2 (because make_moons has x and y coordinates)
**Hidden Layers**: You can set according to your needs
**Output Size**: 1 (for binary classification)

### Network Structure:

- **Layer 1**: Input (2 neurons)
- **Layer 2**: Hidden (variable neurons)
- **Layer 3**: Output (1 neuron)

### Forward Propagation:

```
def __call__(self, x):
    out = x
    for layer in self.layers:
        next_out = []
        for weights, bias in layer:
            activation = bias
            for wi, xi in zip(weights, out):
                activation = activation + wi * xi
            next_out.append(activation.tanh())
        out = next_out[0] if len(next_out) == 1 else next_out
    return out
```

**Step-by-step Process**:

1. Take input values
2. Multiply with weights in each layer
3. Add bias
4. Apply tanh activation function
5. Pass to next layer

[108]

### Mathematical Formula:

```
output = tanh(Σ(weight_i * input_i) + bias)
```

## 4. Make_Moons Dataset - Understanding Data Patterns

```
from sklearn.datasets import make_moons, make_blobs
X, y = make_moons(n_samples=100, noise=0.1)
y = y*2 - 1  # make y be -1 or 1
```

### Dataset Properties:

- **Creates two interleaving half circles**
- **Non-linearly separable data**
- **Perfect for classification problems**
- **Easy to understand through visualization**

[104]

### Why Make_Moons?

1. **Simple**: Only 2D data
2. **Non-linear**: Cannot be separated with a straight line
3. **Visual**: Decision boundary is clearly visible
4. **Testing**: Can test neural network capabilities

### Data Transformation:

```
y = y*2 - 1  # Convert 0,1 to -1,1
```

**Reason**: Neural network output is tanh which gives values from -1 to +1.

## 5. Training Process - How Networks Learn

### Training Loop:

```
learning_rate = 0.001
num_epochs = 400

for epoch in range(num_epochs):
    total_loss = Value(0.0)

    for xi, yi in zip(X, y):
```

```
        inputs = [Value(xi[0]), Value(xi[1])]
        pred = net(inputs)
        target = Value(yi)
        loss = (pred - target) * (pred - target)  # MSE Loss

        # Zero gradients
        for p in net.parameters():
            p.grad = 0.0

        # Backpropagation
        loss.backward()

        # Update parameters
        for p in net.parameters():
            p.data -= learning_rate * p.grad

        total_loss = total_loss + loss
```

[105]

### Training Steps Explanation:

**1. Forward Pass**:

- Put input data through network to get prediction

**2. Loss Calculation**:

- Find difference between predicted and actual values
- **MSE (Mean Squared Error)**: `(predicted - actual)²`

**3. Zero Gradients**:

- Clear gradients from previous iteration
- **Important**: Otherwise gradients will accumulate

**4. Backward Pass**:

- Call `loss.backward()`
- Gradients will be calculated through automatic differentiation

**5. Parameter Update**:

- **Gradient Descent**: `parameter = parameter - learning_rate * gradient`
- **Learning Rate**: How fast to learn

### Hyperparameters:

- **Learning Rate (0.001)**: Slow but stable learning
- **Epochs (400)**: How many times to train on entire dataset
- **Batch Size**: One sample at a time (SGD - Stochastic Gradient Descent)

## 6. Decision Boundary Visualization - Seeing AI Decisions

### Mesh Grid Creation:

```
h = 0.15
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
```

**Purpose**: To create regularly spaced points throughout the 2D space
**Resolution (h=0.15)**: How detailed the boundary should be

[107]

### Predictions on Grid:

```
x_cor = np.c_[xx.ravel(), yy.ravel()]
inputs = [list(map(Value, xrow)) for xrow in x_cor]
predictions = list(map(net, inputs))
```

**Process**:

1. Test neural network on every point in the grid
2. Get prediction for each point
3. Apply threshold for binary classification

### Binary Classification:

```
z = np.array([s.data > 0 for s in predictions])
new_z = z.reshape(xx.shape)
```

**Threshold Logic**:

- If prediction > 0: Class 1 (+1)
- If prediction < 0: Class 2 (-1)

### Visualization:

```
fig = plt.figure()
plt.contourf(xx, yy, new_z, cmap=plt.cm.Spectral, alpha=0.8)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
```

[106]

**Components**:

- **contourf**: Fill decision regions with color
- **scatter**: Show original data points
- **Spectral colormap**: Beautiful color scheme

## 7. Complete Code Workflow

### Step-by-Step Execution:

#### Step 1: Data Preparation

```
X, y = make_moons(n_samples=100, noise=0.1)
y = y*2 - 1  # Convert to -1, +1
```

#### Step 2: Network Initialization

```
net = Neural_Network(2, [16, 16, 1])  # 2 input, 2 hidden layers, 1 output
```

#### Step 3: Training Loop

- Train on data for 400 epochs
- Apply gradient descent for each sample

#### Step 4: Visualization

- Create mesh grid
- Get prediction for each point
- Plot decision boundary

[109]

### Code Flow Diagram:

```
Data Loading → Network Creation → Training Loop → Visualization
     ↓              ↓                ↓              ↓
Make_Moons    Value Objects    Backpropagation   Contour Plot
```

## 8. Practical Insights and Tips

### Why This Implementation is Special:

**1. Custom Automatic Differentiation**:

- Implemented functionality similar to TensorFlow/PyTorch from scratch
- Manually defined gradient for each operation
- Chain rule applies automatically

**2. Educational Value**:

- Understand internal workings of neural networks
- Backpropagation math becomes clear
- Gradient calculation process is visible

**3. Visualization Power**:

- Decision boundary is clearly visible
- Can track training progress
- Understand network behavior

### Common Issues and Solutions:

**Problem 1: Slow Convergence**

- **Solution**: Increase learning rate or change network architecture

**Problem 2: Overfitting**

- **Solution**: Fewer epochs, regularization, or simpler network

**Problem 3: Poor Decision Boundary**

- **Solution**: More hidden neurons, different activation functions

### Performance Tips:

**1. Learning Rate Tuning**:

- Start with 0.01, then experiment
- Too high: Unstable training
- Too low: Very slow learning

**2. Network Architecture**:

- More neurons: Better learning capacity
- More layers: Can capture complex patterns
- Balance between complexity and training time

**3. Data Preprocessing**:

- Normalize input features

- Handle outliers properly
- Ensure sufficient training data

### Real-World Applications:

**1. Image Classification**:

- Same concept, but with CNN architecture
- Softmax output for multiple classes

**2. Natural Language Processing**:

- With word embeddings
- Recurrent networks or Transformers

**3. Recommendation Systems**:

- Learning user-item interactions
- Collaborative filtering

### Conclusion

This implementation is a complete **end-to-end machine learning pipeline** that:

1. **Loads data**
2. **Creates custom neural network**
3. **Uses automatic differentiation**
4. **Trains using gradient descent**
5. **Creates beautiful visualization**

**Key Takeaways**:

- Automatic differentiation is the core of AI
- Neural networks are compositions of mathematical functions
- Visualization helps understand AI decision-making
- Implementation from scratch provides deep understanding

**Future Enhancements**:

- Try different activation functions
- Add regularization techniques
- Test on multiple datasets
- Implement advanced optimizers

This code is just an example - real-world applications use frameworks like PyTorch/TensorFlow, but the fundamental concepts remain the same!